Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies*

Juliana Freire Terrance Swift David S. Warren

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{juliana,tswift,warren}@cs.sunysb.edu

Abstract. Tabled evaluations ensure termination of logic programs with finite models by keeping track of which subgoals have been called. Given several variant subgoals in an evaluation, only the first one encountered will use program clause resolution; the rest uses answer resolution. This use of answer resolution prevents infinite looping which happens in SLD. Given the asynchronicity of answer generation and answer return, tabling systems face an important scheduling choice not present in traditional top-down evaluation: How does the order of returning answers to consuming subgoals affect program efficiency.

This paper investigates alternate scheduling strategies for tabling in a WAM implementation, the SLG-WAM. The original SLG-WAM had a simple mechanism of scheduling answers to be returned to callers which was expensive in terms of trailing and choice point creation. We propose here a more sophisticated scheduling strategy, Batched Scheduling, which reduces the overheads of these operations and provides dramatic space reduction as well as speedups for many programs. We also propose a second strategy, Local Scheduling, which has applications to non-monotonic reasoning and when combined with answer subsumption can improve the performance of some programs by arbitrary amounts.

1 Introduction

Tabling extends the power of logic programming since it allows the computation of recursive queries at the speed of Prolog. This property has led to the use of tabling for new areas of logic programming — not only for deductive database style applications, but other fixpoint-style problems, such as program analysis. Ensuring that these new applications run efficiently in terms of time and space may require the use of different *scheduling strategies*. The possibility of different useful strategies derives from an intrinsic asynchrony in tabling systems between the generation of answers and their return to a given consuming tabled subgoal. Depending on how and when the return of answers is scheduled, different strategies can be formulated. Furthermore, these different strategies can benefit the serious research and industrial applications which are beginning to emerge.

To take a well-known instance, in order to efficiently evaluate queries to disk-resident data, a tabling system should provide set-at-a-time processing analogous to the semi-naive evaluation of a magic-transformed [2] program, so that communication and I/O costs are minimized. To address this, a separate paper defined a breadth-first set-at-a-time strategy for the SLG-WAM [15] of XSB[2] and proved it *iteration equivalent* to the semi-naive evaluation of a magic transformed program [8]. Unlike XSB's original tuple-at-a-time engine, the engine based on the breadth-first strategy showed very good performance for disk accesses.

* To appear in PLILP 96.

[2] XSB is freely available at `http://www.cs.sunysb.edu/~sbprolog`.

UU IR Author Manuscript

Of course tabled evaluations must also be efficient in terms of time and space for in-memory queries. [16] showed that, compared to Prolog, tabled evaluation incurred a minimal execution time overhead under several different criteria of measurement. However, memory is also a critical resource for computations, and stack space for consuming tabled subgoals can be reclaimed only when it can be ensured that all answers have been returned to them. Since the choice of scheduling strategy can influence when this condition happens, it can affect the amount of space needed for a computation.

Finally, a number of tabling applications require more than the simple recursion needed for Horn programs. Resolving a call to a negative literal requires completely evaluating the subgoal contained in the literal, along with all other dependent subgoals. In a similar manner, waiting until part of an evaluation has been completely evaluated can also benefit programs that use *answer subsumption* (e.g. [12]), in which only the most general answers need to be maintained and returned to consuming subgoals. When answer subsumption is seen as taking place over arbitrary lattices (rather than just the lattice of terms), it captures aspects of tabled evaluations for program analyses (see e.g. [4, 10]), for deductive database queries that use aggregates [17], and for answers involving constraints [9].

This paper motivates and describes two new types of scheduling for tabled logic programs:

- We describe Batched Scheduling which is highly efficient for in-memory programs that do not require answer subsumption.
- We describe Local Scheduling, which provides an elegant strategy to evaluate both fixed-order stratified programs, and programs which use answer subsumption.
- We provide detailed results of experiments comparing these two strategies with XSB's original Single Stack Scheduling (described in [16]). They show that:
  - Batched Scheduling can provide an order of magnitude space reduction over the original strategy, as well as reliably provide a significant reduction in time.
  - Local Scheduling can provide large speedups for programs that require answer subsumption, while incurring a relatively small constant cost for programs that do not.

2 Scheduling SLG Evaluations

Review of SLG Terminology

Full details of the concepts and terminology presented in this section can be found in [3]. In an SLG evaluation predicates can be either tabled or non-tabled, in which case SLD resolution is used. Evaluations in tabling systems are usually modeled by a forest of resolution trees containing a tree for every tabled subgoal present in an evaluation. SLG trees have nodes of the form:

$$answer\_template : status : goal\_list$$

(see, e.g. Figures 1 and 3). The *answer_template* is used to maintain variable bindings made to the tabled subgoal during the course of resolution, and the *goal_list* contains unresolved goals. For definite programs, which are the main focus of this paper, *status* is one of the set: *generator*, *active* (or, synonymously, *consuming*), *interior*, *answer*, and *disposed*. The roots of SLG trees are created when new tabled subgoals are selected. These roots have status *generator*, and are also called *generator nodes*. Program clause resolution is used to produce the children of generator nodes; in SLG, this resolution occurs through the SUBGOAL CALL operation. The node calling a tabled subgoal is denoted as a *consuming* or *active* node and its children will be produced by answer clause resolution (through the ANSWER RETURN operation). An answer corresponds to a leaf node of a tree whose *goal_list* is empty. Conceptually, the NEW ANSWER operation can be seen as adding these answers to a table where they are associated with the subgoals that are the roots of their trees. *Interior* nodes represent nodes whose selected literals are *non-tabled* and for which SLD resolution (i.e., program clause resolution in all cases) is used.

Subgoals that occur in trees that have been *completely evaluated* are marked as *disposed*. Sets of subgoals are *completely evaluated* when all possible answers have been derived for them. This concept of *complete evaluation* of a subgoal is necessary for negation and is useful for the early reclamation of stack space in our implementation. When a subgoal is found to be completely evaluated, its non-answer nodes are marked as disposed through the COMPLETION operation.

Any finite SLG evaluation can be seen as a sequence of forests or *SLG systems*. As applicable operations are performed, the evaluation proceeds moving from system to system. Given a particular SLG system, a subgoal S_1 *depends on* a subgoal S_2 iff neither S_1 nor S_2 are completed, and there is a node in the tree for S_1 whose selected literal is S_2. This dependency relation gives rise to a *Subgoal Dependency Graph* (SDG) for each SLG system. Since the dependency relation is non-symmetric, the SDG is a directed graph and can be partitioned into strongly connected components, or SCCs. Note that the dependency graph may be labeled with different types of dependencies: positive or negative dependencies in normal programs, or aggregate dependencies in deductive database programs. Given the usual dependency labelings for normal programs, these SDGs can be related to the dynamic stratification of [13] in the sense that a program is dynamically stratified if there is a dynamic computation rule which avoids labeled SDGs that have cycles through negation. We thus refer to the unlabeled SDG as a *pre-stratification* of a system.

Single Stack Scheduling

The scheduling of resolution in Prolog [1] is conceptually simple. The engine performs forward execution for as long as it possibly can. If it cannot — because of failure of resolution, or because all solutions to the initial query are desired — it checks a scheduling stack (the choice point stack) to determine a *failure continuation* to execute.

The SLG-WAM differs from the WAM in that, in addition to resolving program clauses, it also resolves answers against active nodes. A natural way to extend the WAM paradigm to return answers to an active node is to distinguish between the acts of returning old answers to new active nodes — answers that were already in the table when the active node was created — and returning new answers to old active nodes. The first case is simple: When a new active node is created, a choice point is set up to backtrack through answers in the table much as if they were unit clauses. To handle the second case, whenever a new answer is derived for which there are existing active nodes, an answer return choice point is placed on the choice point stack. This choice point will manage the resolution of the new answer with each of the active nodes. Forward execution is then continued until failure, at which time the top of the choice point stack is then used for scheduling. The choice point stack thus serves as a scheduling stack for both returning answers and resolving program clauses. Accordingly, we call this scheduling strategy Single Stack Scheduling. The operational semantics of this scheduling strategy was described in detail in [14] and forms the basis of XSB's SLG-WAM as described in [15]. The following example demonstrates how this strategy works.

Example 1. Consider the following double-recursive transitive closure

```
:- table p/2.
a(1,2). a(1,3). a(2,3).
p(X,Y) :- p(X,Z), p(Z,Y).
p(X,Y) :- a(X,Y).
```

and the query `?- p(1,Y)`. The forest of SLG trees for this program at the end of the evaluation is shown in Fig. 1, and Fig. 2 shows snapshots of the choice point stack at different points of the evaluation.

Let us examine the actions of Single Stack Scheduling in detail. When the top-level query is called, it is inserted into the table (since `p/2` is a tabled subgoal) and a generator choice point is

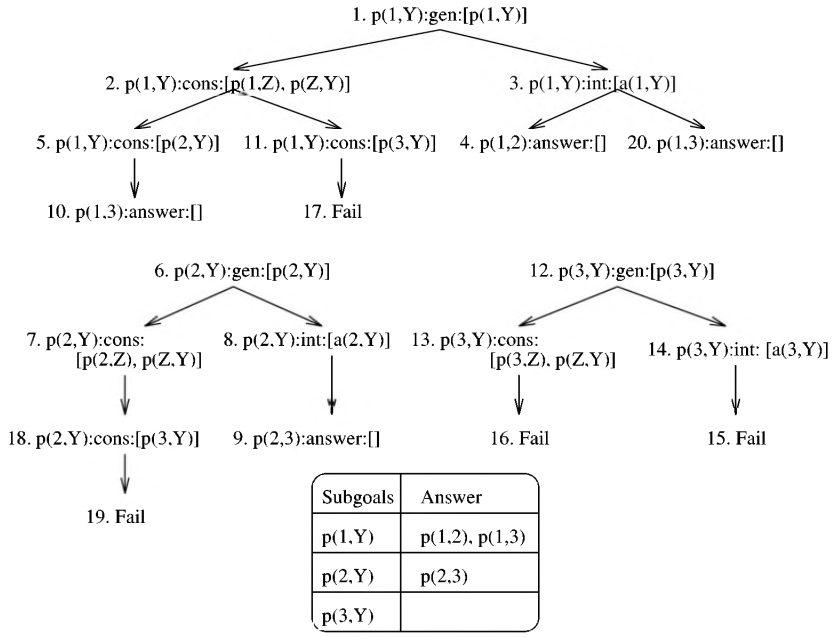


Fig. 1. *SLG evaluation under Single Stack Scheduling*

created, which corresponds to the root of the new SLG tree in node 1. Program clause resolution is then used to create node 2. Since the selected literal in node 2 is a variant of a tabled subgoal, a new active choice point frame (corresponding to the consuming node 2) is laid down to serve as an environment through which to return answers, and the stacks are frozen (see Fig. 2(a)), so that backtracking will not *overwrite* any frames below that point. In addition, if there were any answers in the table, the RetryActive instruction would backtrack through them and return each to the active node. Since there are no answers in the table for `p(1,Y)`, the second clause for `p/2` is tried. As the selected literal is not a tabled predicate, SLD resolution is applied and a Prolog choice point is laid down for `a/2`. The evaluation then gives rise to an answer, `p(1,2)` (in node 4). Since there are no variants of `p(1,2)` associated with `p(1,Y)`, the answer is inserted into the table, and an answer return choice point is laid down (Fig. 2(a)).

When the engine backtracks into the answer return choice point for `p(1,2)` (see Fig. 2(b)), the AnswerReturn instruction freezes the stacks and returns the answer to all active nodes[3]. In this case, the answer will be returned to node 2, which in turn will trigger a call to `p(2,Y)`. The evaluation of `p(2,Y)` (see node 6) is similar to that of node 1: It is inserted into the table and a generator choice point is laid down for it. It will eventually generate an answer (`p(2,3)` in node 9), which is inserted into the table for `p(2,Y)`. In addition, an answer return choice point is laid down and the bindings for the answer are propagated to node 5, which will then derive the answer `p(1,3)`. When `p(1,3)` is returned to node 2, it prompts a call to `p(3,Y)`.

A generator and a new active choice point are laid down for `p(3,Y)`. But this subgoal fails, and can be completed. Upon completion, the choice points for `p(3,Y)` can be reclaimed — as Fig. 2(c) shows. At this point, the engine backtracks into the answer return choice point `p(2,3)`, and this answer is returned to the consuming node 7. Node 7 then calls `p(3,Y)`, which is completed and has no answers, and thus the engine fails. The subgoal `p(2,Y)` is now completely evaluated, and space

[3] As an optimization, in the SLG-WAM after an answer return choice point returns the answer to the last active node, it removes itself from the backtracking chain — this is represented in Fig. 2 by dotted frames.

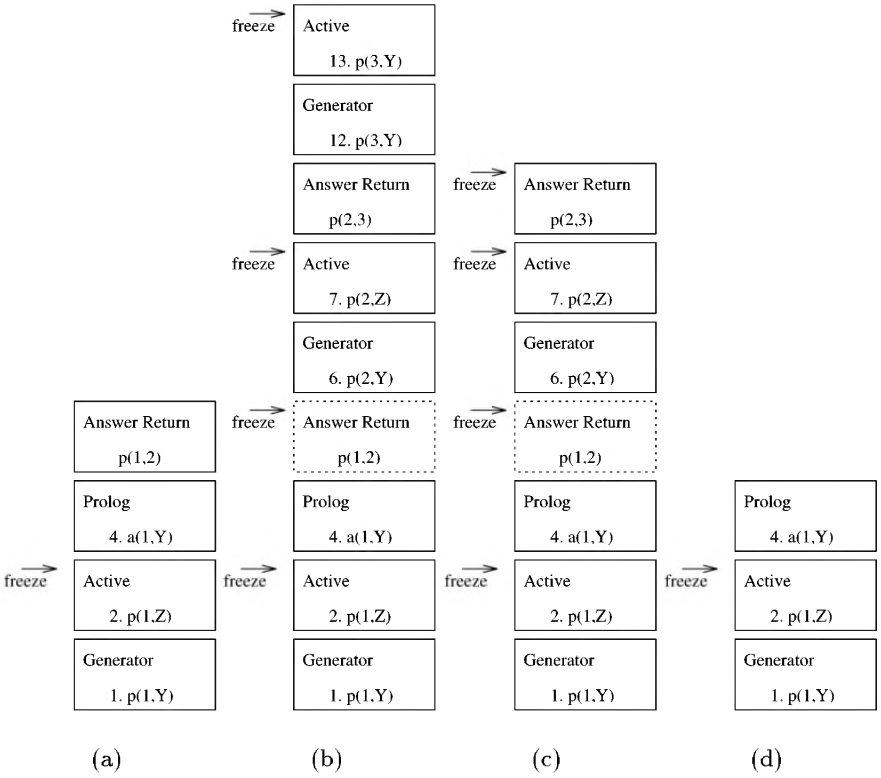


Fig. 2. *Snapshots of the completion stack during the evaluation of the program in Example 1 under Single Stack Scheduling*

for it can be reclaimed in the stacks (see Fig. 2(d)).

The evaluation then returns to the choice point for `a/2` (node 3), and the next clause is tried. The answer `p(1,3)` is generated (node 20), but since a variant of this answer is already in the table (from node 10), the engine simply fails. Finally, when the engine backtracks to the generator node for `p(1,Y)` (node 1) and there are no other choices to be tried, the last subgoal in the system can be completed.

While single-stack scheduling is simple to conceptualize, it contains several drawbacks. First, the integration of the action of returning answers into the mechanism of the choice point stack makes Single Stack Scheduling not easily adaptable to a parallel engine [6]. Another problem is memory usage: In order to perform answer clause resolution at different points in the SLG forest, the stacks have to be frozen so that the environment can be correctly reconstructed at the different points. This need to freeze stacks may lead to inefficient space usage by the SLG-WAM, as some frames might get trapped (e.g., the Prolog choice point in Fig. 2(b)). Finally, the addition of new choice points and the need to move around in the SLG forest to return answers means that trailed variables must be continually set and reset to switch binding environments, causing further inefficiencies.

3 Batched Scheduling

Batched Scheduling can be seen as an attempt to address the problems with Single Stack Scheduling mentioned above. Indeed, versions 1.5 and higher of XSB use this new strategy as a default. As its

name implies, Batched Scheduling minimizes the need to freeze and move around the search tree simply by *batching* the return of answers. That is, if the engine generates answers while evaluating a particular subgoal, they are added to the table and the subgoal continues its normal evaluation until it resolves all available program clauses. Only then will it return the answers it generated during the evaluation to consuming nodes. As demonstrated in Section 5, this new strategy makes better use of space: By reducing the need to freeze branches it reduces the number of trapped nodes in the search tree. Along with reducing space, Batched Scheduling shows significantly better execution times. The following example illustrates some of the differences between Single Stack Scheduling and Batched Scheduling.

Example 2. The execution of the program and query from Example 1 under Batched Scheduling is depicted through the SLG forest in Fig. 3 and a sequence of choice point stacks in Fig. 4. As can be seen from comparing the forests in Fig. 1 and Fig. 3, the procedures are identical through the first four resolution steps, but differ in the fifth step. Here, Batched Scheduling resolves a program clause against node 3 while Single Stack Scheduling returns the answer `p(1,2)` that was derived in step 4 to node 2. This difference reflects two of the problems mentioned above. First, Single Stack Scheduling requires an environment switch from node 3 to node 2 to return the answer, and will later require a swich *back* to node 3 to finish program clause expansion for that node. Furthermore, the unexpanded program clause for node 3 is stored in the engine as a choice point. This choice point not only takes up space itself, but the need to later switch back to it requires freezing (at AnswerReturn) the stack at that choice point (see Fig. 2(b)). This frozen space cannot be reclaimed until completion of the SCC in which it lies.

A similar difference occurs when the answer `p(2,3)` is returned to the consuming node 6. In the choice point stack of Fig. 2(c), the return of this answer requires the placement of an explicit choice point and a freeze. In Fig. 4 both of these overheads are avoided.

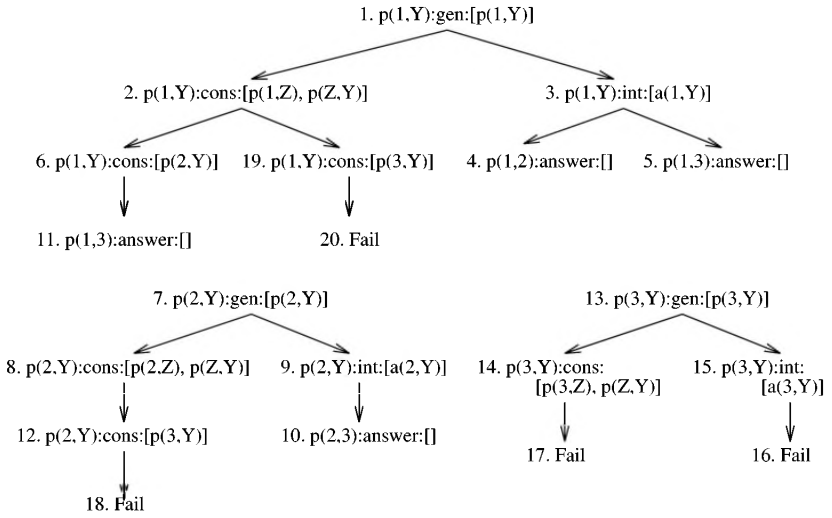


Fig. 3. *SLG evaluation under Batched Scheduling*

Rather than creating explicit answer return choice points, as in Single Stack Scheduling, Batched Scheduling controls answer return through a designated node in each SCC called a *leader*[4]. When the

[4] In the SLG-WAM each subgoal is labeled with a unique depth-first number (DFN) reflecting its location in the stack, which maintains the exact order in which subgoals are called. The leader of an SCC is the subgoal in that SCC with the lowest DFN (i.e., the subgoal in the SCC that was first called).

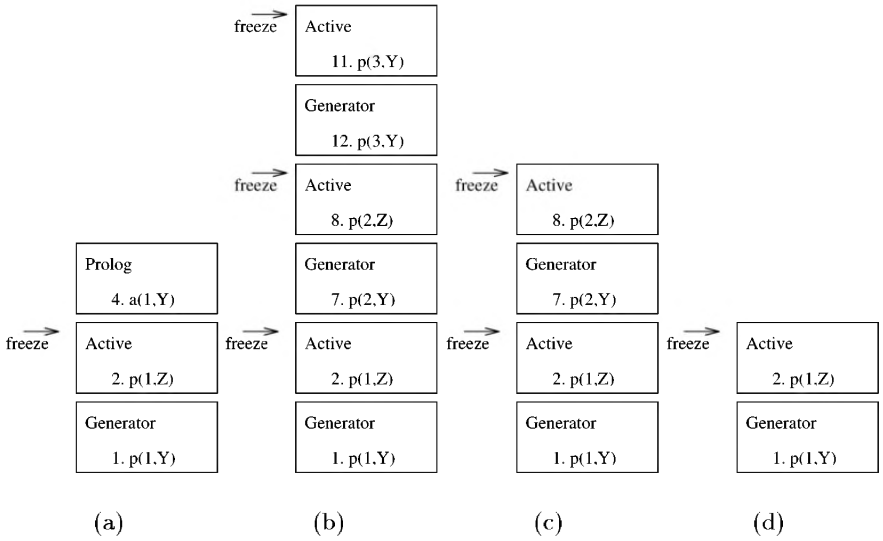


Fig. 4. *Snapshots of the completion stack during the evaluation of the program in Example 1 under Batched Scheduling*

generator choice point has exhausted all program clause resolution, rather than disposing of the choice point, as in Prolog, the failure continuation instruction for the choice point becomes a CheckComplete instruction. If the generator choice point corresponds to a subgoal that is a leader of an SCC, the action of this instruction is to cycle through the subgoals in the SCC to return unresolved answers to every consuming node whose selected literal is in the SCC. Batched Scheduling is related to fixpoint style algorithms of deductive databases. The engine iteratively backtracks to the CheckComplete instruction for the leader of an SCC. This leader then switches environments to a consuming node, *Cons*, that has a non-empty delta set of answers. (If there is no such *Cons*, the fixpoint for the SCC has been reached and the subgoals in the SCC can be completed). Each new answer for *Cons* is resolved through backtracking as with unit clauses (as in the RetryActive instruction of Single Stack Scheduling). Furthermore, the engine resolves answers against *Cons* as long as there are any answers to resolve, and may resolve answers in the same iteration they are added. This latter step gives good performance for in memory queries, but makes the Batched Scheduling algorithm differ from traditional goal-oriented formulations of semi-naive fixpoint[5] such as Magic evaluation [2].

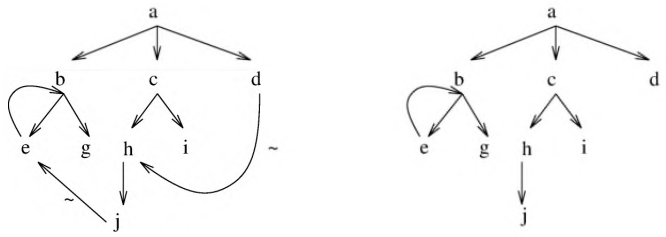
Batched Scheduling bears some resemblance to other two independently developed approaches: the ET* algorithm from [5] and the AMAI from [10]. However, in [5], Fan and Dietrich do not consider strongly connected components in the fixpoint check, and their strategy is *fair* for answers, what might cause inefficiencies. In [10], Janssens et al. describe an abstract machine specialized for abstract interpretation and use a similar scheduling strategy for their fixpoint iterations. Even though they take SCCs into account, these are detected statically, whereas in our engine in order to evaluate general logic programs (with negation), SCCs have to be determined at run time.

4 Local Scheduling

In Local Scheduling the engine evaluates a single SCC at a time. In other words, answers are returned outside of an SCC (that is, to consuming nodes in trees whose root subgoal is not in the SCC) only

[5] Details of these differences can be found in [7].

after that SCC is completely evaluated. The action of Local Scheduling can easily be seen through the following example of stratified evaluation.



(a) Negative dependencies (b) No negative dependencies

Fig. 5. *Subgoal dependency graphs for program* P *of Example 3 under different search strategies*

Example 3. Let P be the modularly stratified program

```
:- table a/0,b/0,c/0,d/0,e/0,g/0,h/0,i/0,j/0.

a:-b,c,d.      b:-e.          c:-h.
               b:-g.          c:-i.

d:- ~h.        e:-b,s.        g.

h:-j.          j:- ~e.        i.
```

for which the query `?- a` is to be evaluated. If evaluated under either Single Stack Scheduling or Batched Scheduling, an SDG will be produced with cascading negative dependencies as shown in Fig. 5(a). Even though there is no cycle through negation, detecting this property can complicate the evaluation of stratified programs. However if Local Scheduling is used, a *simpler* SDG is created, as depicted in Fig. 5(b). To attain this latter SDG, each independent SCC is completely evaluated before returning any answers to subgoals outside the SCC — making the search depth-first with respect to SCCs. In Local Scheduling, the SCCs {`b, e`} and {`g`} are completely evaluated before `b` returns any answers to `a`. Thus, `e` is completely evaluated when ~`e` is called and negative dependencies are not created. Both the negative link from `j` to `e`, and that from `d` to `h` are avoided, since both `e` and `h` are completed by the time they are called negatively.

Local Scheduling can improve the performance of programs that benefit from answer subsumption in the following manner. Answer subsumption can be performed as a variation of the SLG NEW ANSWER operation. While adding an answer, the engine may check whether that answer is more general than those currently in the table. If it is more general, this new answer is added and the subsumed answers are removed. Given that Local Scheduling evaluates each SCC completely before returning any answers out of it, we are guaranteed that only the most general answers will be returned out of that SCC. This process is presented in detail in Example 4.

Example 4. Consider the following variation of the same generation program which finds the smallest distance between two people in the same generation

```
sgi(X,Y)(I) :-
      ancestor(X,Z),
      subsumes(min)(sgi(Z,Z1),I1),
      ancestor(Y,Z1),
      I is I1+1.
sgi(X,X)(0).
:- subsumes(min)(sgi(joan,carl),I).
```

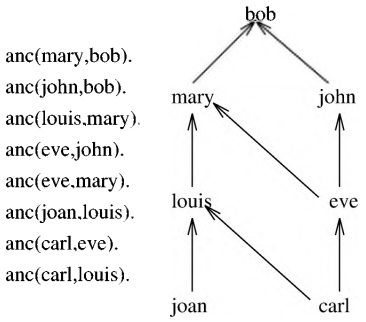


Fig. 6. *Ancestor relation for Example 4*

where `subsumes(min)/2` is a tabled predicate that performs answer subsumption by deleting all non-minimal answers every time it adds an answer to the table. Given the facts in Fig. 6, there are a number of ways this query can be evaluated. It is well-known that for shortest-path like problems, a breadth-first search can behave asymptotically better than depth-first. Nevertheless, in this example a breadth-first search is still not *optimal*.

In the above query we are trying to find how close `joan` and `carl` are. Note that they have three common ancestors (`louis`, `mary` and `bob`), and thus they are cousins of 1st, 2nd and 3rd degree. If evaluated under breadth-first (the behavior of Batched Scheduling for this example), all possible subpaths between `joan` and `carl` are considered, and if at some point during the evaluation a subpath is found whose length is less than those so far derived, it is immediately propagated even though it may not be minimal. For instance, Batched Scheduling first finds the distance between two immediate ancestors of `joan` and `carl` to be 2, and concludes the distance between `joan` and `carl` themselves is 3. Then evaluation continues and a new distance between the immediate ancestors if found to be 1, a new answer (`I=2`) is generated for the top-level query. Finally, the minimal distance between `joan` and `carl` is found to be 1 and the *correct* answer is returned.

If Local Scheduling is used instead, only minimal subpaths are propagated, and the engine is able to prune a number of superfluous choices.

5 Experimental Results

Due to space limitations details on the implementations of the strategies have been omited[6]. In this section we compare both execution time and memory usage of SLG-WAM engines based on the different scheduling strategies described in the paper. XSB v. 1.4 uses Single Stack Scheduling, XSB v. 1.5 uses Batched Scheduling and XSB Local uses Local Scheduling. For execution time, we considered not only the running time, but also the dynamic count of SLG-WAM instructions and operations. Benches were run on a SPARC2 with 64MB RAM under SUNOS.

The bench programs consisted of variations of transitive closure, same-generation and shortest-path on various graphs (the programs are described in Table 1). We experimented with graphs that have well defined structures, such as linear chains and complete binary trees, as well as less regular graphs, based on Knuth's *Words*[7].

[6] Implementation details are given in the expanded version of this paper available at `http://www.cs.sunysb.edu/~sbprolog`.

[7] The nodes of this graph are the 5757 more common 5-letter English words; there is an arc between two words if they differ in a single character [11].

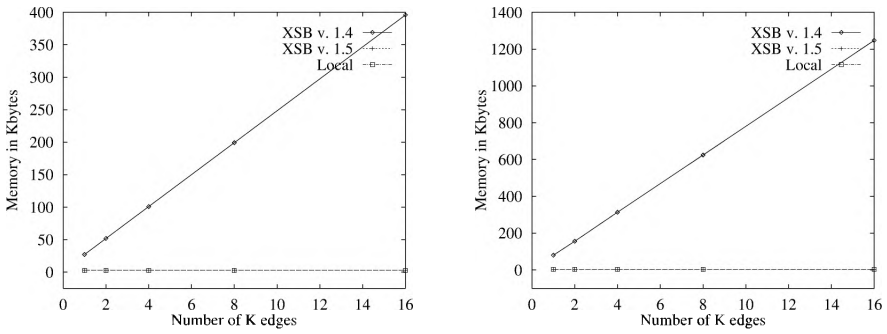
Transitive Closure	reach(X,Y) :- arc(X,Y). reach(X,Y) :- reach(X,Z), arc(Z,Y).
Shortest-Path	sp(X,Y)(D) :- arc(X,Y,D). sp(X,Y)(D) :- subsumes(min)(sp(X,Z),D1), arc(Z,Y,D2), D is D1+D2.
Same Generation	sgi(X,Y)(D) :- arc(X,Y). sgi(X,Y)(D) :- arc(X,Z), subsumes(min)(sgi(Z,Z1),D1), arc(Y,Z1), D is D1+1.

Table 1. *Bench programs*

5.1 Performance of Batched Scheduling

Let us first examine the differences between Single Stack Scheduling and Batched Scheduling for left-recursive transitive closure on a linear chain containing 1024 nodes, with the query `reach(1,X)`. Under Single Stack Scheduling, first all facts are used (by backtracking through the facts for `arc/2` in the first clause) and when the active node is laid down for the subgoal in the second clause, each answer in the table is consumed. If a new answer is derived in this process, computation is suspended and the new answer is immediately returned, by freezing the stacks and pushing an answer return choice point onto the choice point stack. Under Batched Scheduling strategy, all answers in the table are returned before any newly derived answer is considered.

We profiled the SLG-WAM instructions and the main difference between XSB v. 1.4 and XSB v. 1.5 for this example lies in the fact that since answer return choice points are no longer used, AnswerReturns are replaced by RetryActives. Since RetryActive requires fewer (about 30% less) machine instructions than AnswerReturn, the tradeoff is beneficial. Also, since Batched Scheduling requires less stack freezing, it utilizes memory better. Fig. 7(a) gives the total stack space usage (local, global, choice point, trail, and completion stack) for the three strategies for the same left-recursive transitive closure and query on chains of varying lengths. Note that whereas memory consumption grows linearly with the number of facts for XSB v. 1.4, the space remains constant for XSB v. 1.5 (and also for Local) at 2.7 Kbytes.



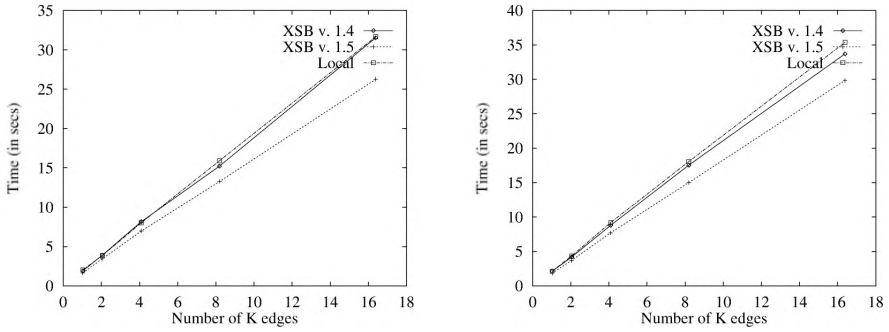
(a) Linear chains of varying length

(b) Complete binary trees of varying height

Fig. 7. *Total memory usage for left-recursive transitive closure*

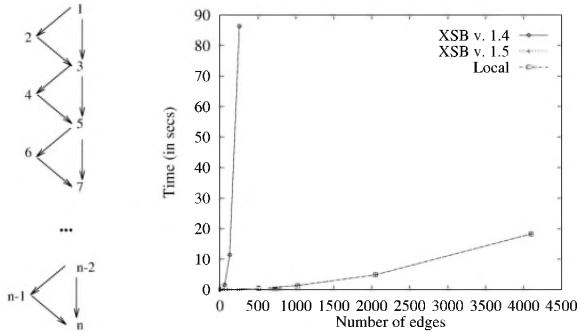
The SLG-WAM instruction count for left-recursive transitive closure on complete binary trees of varying height are similar to those for chains. However, the batching of answer resolution reduces the need for the engine to move around in the SLG forest and thus Batched Scheduling also save trails and untrails. As a result, memory savings are even bigger than for chains, as Fig. 7(b) shows (note that for the two new strategies the space remains constant at 2.88Kbytes).

The times for the different engines to compute the transitive closure on trees and chains is given in Fig. 8. The speedup of XSB v. 1.5 over XSB v. 1.4 for these examples varies between 11 and 16%.



(a) Linear chains of varying length (b) Complete binary trees of varying height

Fig. 8. *Times for left-recursive transitive closure*



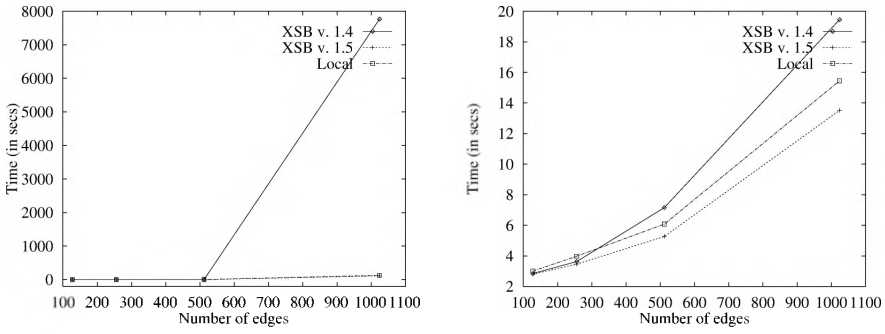
(a) (b)

Fig. 9. *(b) Shows the time in secs. for XSB v. 1.4, XSB v. 1.5 and Local to find the shortest-path between the endpoints (*1 *and* n*) of a graph of the form depicted in (a)*

Single Stack Scheduling as its name implies, uses a stack-based scheduling for answers, and so when executing transitive closure over, say a binary tree, traverses the tree in a depth-first manner. Because Batched Scheduling effectively uses a queue for returning answers, when executing (left-recursive) transitive closure it will traverse the same tree in a breadth-first manner. Accordingly, optimization problems such as shortest-path that can (1) be formulated through left-recursive transitive closure, and (2) benefit from a breadth-first search, can be run more efficiently under Batched Scheduling[8].

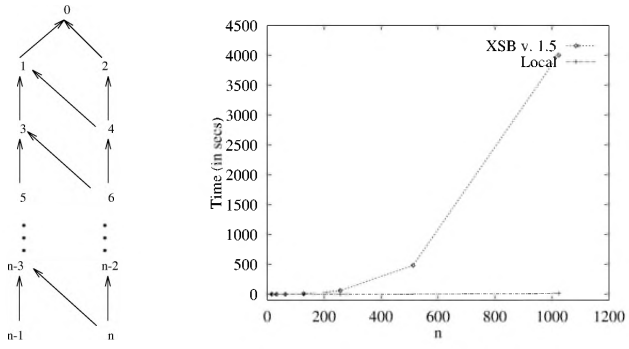
[8] It is worth pointing out that only the underlying data structures are searched in a breadth-first manner. The depth-first nature of program clause resolution in the WAM is maintained through all strategies discussed in this paper.

To demonstrate this, we compared the running times for the different engines using the shortest-path program in Table 1. First we considered the artificial graph shown in Fig. 9(a). If a depth-first search is used to compute the shortest-path between nodes 1 and `n` in this graph, it will run in exponential time. However the shortest path can be computed in polynomial time if the graph is searched in a breadth-first manner. Fig. 9(b) shows the times XSB v. 1.4, XSB v. 1.5 and Local take to compute `sp(1,n)(Dist)` for different values of `n`. In addition to running slower, XSB v. 1.4 ran out of memory on graphs with more than 512 nodes. We also considered more realistic graphs, variations of Knuth's Words, and the speedups are substantial as Fig. 10 shows.



(a) `sp(words,X)(Dist)` – *500 iterations* (b) `sp(there,white)(Dist)` – *1000 iterations*

Fig. 10. *Timings for shortest-path on Words*



(a) (b)

Fig. 11. *(b) shows the execution time for the query* `subsumes(min)(sgi(n-1,n),I)` *on graphs of the form depicted in (a) for varying* `n`

5.2 Performance of Local Scheduling

Since the return of answers out of an SCC has to be delayed until the SCC is completely evaluated, the implementation of Local Scheduling incurs the cost of explicitly returning an answer to the generator node, rather than sharing the bindings as in Batched Scheduling and Single Stack Scheduling. This

results in the duplication of the number of RetryActive instructions and in a higher number of environment switches.

As for memory consumption, Local Scheduling has the same constant behavior as Batched Scheduling for transitive closure on trees and chains, as evidenced in Fig. 7 (notice that the lines for Local and XSB v. 1.5 overlap). In Fig. 8 we can see that Local adds a roughly constant 15% overhead to XSB v. 1.5, and the execution times for the Local engine are comparable to XSB v. 1.4. Local also has approximately the same performance as XSB v. 1.5 for shortest-path on variations of the Words graph and on the artificial graph of Fig. 9(a) (see Fig. 9(b) and Fig. 10).

We have stated that for programs that can benefit from answer subsumption Local Scheduling can perform arbitrarily better than Batched Scheduling. The graph in Fig. 11(b) substantiates this statement. This experiment measured the times to find the shortest distance between the two *deepest* nodes (`n-1` and `n`) on graphs of the form depicted in Fig. 11(a) for varying `n`, using the same-generation program of Example 4. Note that the times for the Local engine vary from 0.06 to 15.7 seconds, whereas for XSB v. 1.5, they range between 0.09 and 4007.8 seconds.

6 Conclusion

This paper proposes new scheduling strategies that can improve the performance — memory usage and execution time – of tabled evaluations. Due to its performance improvement, Batched Scheduling is now the default scheduling strategy for XSB. The gains from this strategy are twofold: by eliminating the answer return choice point and the freezing of stacks done at AnswerReturn, memory usage is greatly reduced; and because of the reduction of trailings/untrailings, the execution time decreases.

Local Scheduling can perform asymptotically better than Batched Scheduling when combined with answer subsumption. This can be of use in many different areas such as aggregate selection and program analysis. In addition, Local Scheduling may have an important role to play in evaluating programs under the well-founded semantics [18]. Currently in XSB the engine may have to construct part of the SDG to check for loops through negation. Since Local Scheduling maintains exact SCCs, it does not require this step as was demonstrated by Example 3. Furthermore, when negative literals actually are involved in a loop through negation, SLG uses a DELAY operation to attempt to break the loop. This use of DELAY may create an answer A that is *conditional* on the truth of some unevaluated literal. However, other derivation paths may create an *unconditional* answer for A (for example, all answers considered in this paper are unconditional). Clearly conditional answers are not needed for A if there is a corresponding unconditional answer, and the use of DELAY gives rise to a form of answer subsumption, leading to another advantage of locality. As the well-founded semantics becomes used by practical programs, the advantages of Local Scheduling may become increasingly necessary for their efficient evaluation.

Acknowledgements: This work was supported in part by CAPES-Brazil, and NSF grants CDA-9303181 and CCR-9404921.

References

1. H. Aït-Kaci. *WAM: A Tutorial Reconstruction.* MIT Press, 1991.
2. C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3):255–299, 1991.
3. W. Chen and D.S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *JACM*, 43(1):20–74, January 1996.

4. S. Dawson, C.R. Ramakrishnan, and D.S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 117–125. ACM, 1996.
5. C. Fan and S. Dietrich. Extension Table Built-ins for Prolo. *Software–Practice and Experience*, 22(7):573–597, July 1992.
6. J. Freire, R. Hu, T. Swift, and D.S. Warren. Exploiting Parallelism in Tabled Evaluations. In *7th International Symposium, PLILP 95 – LNCS Vol. 982.* Springer-Verlag, 1995.
7. J. Freire, T. Swift, and D.S. Warren. Batched answers: An alternative strategy for tabled evaluations. Technical Report 96/2, Department of Computer Science, State University of New York at Stony Brook, 1996.
8. J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. Technical Report 96/4, Department of Computer Science, State University of New York at Stony Brook, 1996.
9. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 111–119, 1987.
10. G. Janssens, M. Bruynooghe, and V. Dumortier. A Blueprint for an Abstract Machine for Abstract Interpretation of (Constraint) Logic Programs. In *Proceedings of the International Symposium on Logic Programming (ILPS)*, 1995.
11. D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison Wesley, 1993.
12. G. Köstler, W. Kiessling, H. Thöne, and U. Güntzer. Fixpoint iteration with subsumption in deductive databases. *Journal of Intelligent Information Systems (JIIS)*, 4(2):123–148, March 1995.
13. T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the ACM Symposium on Principle of Database Systems (PODS)*, pages 11–21, 1989.
14. T. Swift. *Efficient Evaluation of Normal Logic Programs.* PhD thesis, Department of Computer Science, State University of New York at Stony Brook, 1994.
15. T. Swift and D. S. Warren. An Abstract Machine for SLG Resolution: Definite Programs. In *Proceedings of the International Symposium on Logic Programming (ILPS)*, pages 633–654, 1994.
16. T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *Proceedings of the International Symposium on Logic Programming (ILPS)*, pages 219–238, 1994.
17. A. van Gelder. Foundations of Aggregation in Deductive Databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 13–34, 1993.
18. A. van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

UU IR Author Manuscript